A Theory of Contexts for Higher-Order Encodings of Process Algebras

Ivan Scagnetto

ivan.scagnetto@uniud.it

Department of Mathematics, Computer Science, and Physics (University of Udine)

BEHAPI 2019 Workshop on Behavioural APIs
Prague (Czech Republic), 06/04/2019

Introduction

- Process algebras/calculi are complex formal systems, representing typical case studies in *Computer Aided Formal Reasoning* (CAFR).
- Ultimate goal: **rigorous certification** of concurrent programs properties (proofs are long and intricate: errors easily ensue with pencil and paper).
- The main alternatives are:
  - developing **ad-hoc** tools for every object language;
  - encoding object languages in a **Logical Framework (LF)**, i.e, a metalogical formalism already featuring many intrinsic aspects and notions of a wide range of formal systems.
- Encoding a formal system in a LF is like writing programs in a **general-purpose** programming language.

Type Theory based Logical Frameworks

A type theory based LF provides the means to represent many logical and formal systems.

Main features:

- the underlying metalanguage is a typed λ-calculus with dependent types;
- **Curry-Howard isomorphism**: types are interpreted as judgments and terms are viewed as proofs of the judgment corresponding to their type (**proof checking** $\rightsquigarrow$ **type checking**);
- the basic judgments family is extended with two higher-order forms:

$$\overline{J_1 \vdash J_2} = \overline{J_1} \rightarrow \overline{J_2} \qquad \text{(logical consequence)}$$

  and

$$\overline{\bigwedge_{x \in C} J(x)} = \Pi x : \overline{C}.\overline{J(x)} \qquad \text{(schematic judgment)}$$

  where $\overline{C}$ is the type representing the syntactic category C.
- an object language is represented by a signature (i.e., a set of typed constants): the encoding is adequate iff there exists a **compositional bijection** between the object language entitites and the canonical forms of the LF.

Syntax representation: variables and binders (I)

de Bruijn indices [DB72]: variables are represented by indices (natural numbers); good for implementations, but not for an interactive proof assistant (the encoding is very difficult to read back). There is not a notion of α-conversion (there are no variables) and capture-avoiding substitution can be automatized, paying the price to formalize and prove a large number of lemmas about the handling of indices.

First-order abstract syntax [GM96]: variables (both free and bound) are represented as terms of a suitable type `var`/`name`. Binders are rendered in a “flat” way, forcing the user to explicitly encode and deal with α-conversion and capture-avoiding substitution.

Locally Nameless [Cha12]: first-order abstract syntax for free variables, de Bruijn indices for bound variables.

Nested abstract syntax [HM12]: free variables are represented by a suitable type V, whereas binders and bound variables are rendered using a nested datatype on V to denote “fresh” elements.

Syntax representation: variables and binders (II)

Higher-order abstract syntax (HOAS [PE88]): HOAS represents binders as *functional constants* and variables/names as metavariables of the same type of terms, thus delegating α-conversion and capture-avoiding substitution to the metalanguage.

Weak Higher-order abstract syntax (wHOAS [DFH95]): wHOAS represents binders as *functional constants* and variables/names as a distinct type, delegating α-conversion and renamings to the metalanguage, but requiring an explicit representation of capture-avoiding substitution of terms for variables.

Nominal representation [GP99]: the encoding is close to the original object language, but α-conversion and capture-avoiding substitution must be expressed in terms of variable/name permutations.

Untyped λ-calculus

First-order abstract syntax:

$$\begin{array}{lcl} var & : & Set \\ tm & ::= & is_{var} : var \to tm \\ & \mid & app : tm \to tm \to tm \\ & \mid & lam : var \to tm \to tm \end{array}$$

The type of *lam* does not denote it as a binder.

Higher-order abstract syntax:

$$\begin{array}{lcl} tm & ::= & app : tm \to tm \to tm \\ & \mid & lam : (tm \to tm) \to tm \end{array}$$

- Variables of the untyped λ-calculus are rendered by the framework metavariables of type *tm*.
- The functional type of *lam* allows us to use the metalanguage binder λ to render the object language abstraction:

$$\epsilon_\emptyset(\lambda x.x) \ = \ lam(\lambda x : tm.\ x)$$

HOAS-based encodings: pros and cons

♡ α-conversion and *capture-avoiding substitution* are **delegated** to the metalanguage. Substitution is rendered by functional application:

$$\text{Example } (\beta\text{-reduction}): \; app(lam(f), t) \text{ reduces to } f(t)$$

♠ **Expressivity**: constructors with type $(A \to A) \to A$ force to give up with (co)inductive features; otherwise, non-normalizing terms arise:

$$F \stackrel{\text{def}}{=} (\lambda x{:}tm.\ Case\ x\ of\ \lambda x, y{:}tm.\ (app\ x\ y)\ \lambda f{:}tm \to tm.\ f(lam\ f)\ end) : tm \to tm$$

$$F(lam\ F) : tm \to_\beta F(lam\ F) : tm \to_\beta \ldots$$

♠ **Exotic terms**: *higher-order* constructors, e.g. $c : (A \to B) \to B$ (with $A, B : Set$ inductive), yield terms of the form $(c\ \lambda x : A.\ (Case\ x\ of\ \ \ldots\ end))$ which do not correspond to any objects of the encoded system.

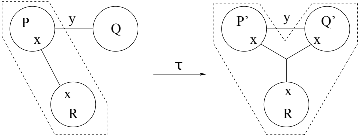
♠ Developing the metatheory of the object language may be very difficult:

1. we lack a higher-order induction principle for reasoning about the structure of functional terms;
2. the "freshness" information about bound variables is not available during proofs;
3. "trivial" properties (e.g., if $P \equiv Q$ and $x \notin FV(P, Q)$, then $P\{x/y\} \equiv Q\{x/y\}$), usually taken for granted with pencil and paper, are not provable.

An example encoding: the π-calculus

Milner, Parrow, Walker [MPW92a, MPW92b]

Well known **Process Algebra** standing as paradigm of concurrent programming. It allows one to model environments of processes endowed with a mobile topology of communication channels:



Components:

syntax of *names* ($\mathcal{N}$), *actions* and *agents* (*processes*, $\mathcal{P}$);

operational semantics, i.e., labelled transition relation: $\xrightarrow{\alpha} \subseteq \mathcal{P} \times \mathcal{P}$;

equivalence relation between processes: $\dot{\sim} \subseteq \mathcal{P} \times \mathcal{P}$.

π-calculus syntax

Agents/Processes

$$
\begin{aligned}
P ::= \quad & 0 \mid \bar{x}y.P \mid x(y).P \mid \tau.P \mid (\nu x)P \mid !P \\
& \mid P_1 | P_2 \mid P_1 + P_2 \mid [x = y]P \mid [x \neq y]P
\end{aligned}
$$

Actions

α	Type	$fn(\alpha)$	$bn(\alpha)$
τ	Free	$\emptyset$	$\emptyset$
$\bar{x}y$	Free	$\{x, y\}$	$\emptyset$
$x(y)$	Bound	$\{x\}$	$\{y\}$
$\bar{x}(y)$	Bound	$\{x\}$	$\{y\}$

Operational semantics of π-calculus

$$\text{OUT}\ \frac{-}{\overline{x}y.P \xrightarrow{\overline{x}y} P} \qquad \text{IN}\ \frac{-}{x(z).P \xrightarrow{x(w)} P\{w/z\}}\ w \notin fn((\nu z)P)$$

$$\text{SUM}_l\ \frac{P \xrightarrow{\alpha} P'}{P+Q \xrightarrow{\alpha} P'} \qquad \text{PAR}_l\ \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q}\ bn(\alpha)\cap fn(Q)=\emptyset$$

$$\text{COM}_l\ \frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{y/z\}} \qquad \text{RES}\ \frac{P \xrightarrow{\alpha} P'}{(y)P \xrightarrow{\alpha} (y)P'}\ y \notin n(\alpha)$$

$$\text{MATCH}\ \frac{P \xrightarrow{\alpha} P'}{[x=x]P \xrightarrow{\alpha} P'} \qquad \text{MISMATCH}\ \frac{P \xrightarrow{\alpha} P'}{[x\neq y]P \xrightarrow{\alpha} P'}\ x \neq y$$

$$\text{OPEN}\ \frac{P \xrightarrow{\overline{x}y} P'}{(\nu y)P \xrightarrow{\overline{x}(w)} P'\{w/y\}}\ \begin{array}{l} y \neq x \\ w \notin fn((\nu y)P') \end{array} \qquad \text{TAU}\ \frac{-}{\tau.P \xrightarrow{\tau} P}$$

$$\text{CLOSE}_l\ \frac{P \xrightarrow{\overline{x}(w)} P' \quad Q \xrightarrow{x(w)} Q'}{P|Q \xrightarrow{\tau} (\nu w)(P'|Q')} \qquad \text{REPL}\ \frac{P|!P \xrightarrow{\alpha} P'}{!P \xrightarrow{\alpha} P'}$$

Strong (late) bisimilarity

Definition: a binary relation $\mathcal{S}$ on processes is a *strong simulation* iff, for every pair of processes P, Q, if $P \mathrel{\mathcal{S}} Q$ then

1. if $P \xrightarrow{\alpha} P'$ and α is a free action, then there $\exists Q'.Q \xrightarrow{\alpha} Q'$ and $P' \mathrel{\mathcal{S}} Q'$;
2. if $P \xrightarrow{x(y)} P'$ and $y \notin n(P, Q)$, then there $\exists Q'.Q \xrightarrow{x(y)} Q'$ and for all $w \in \mathcal{N}$: $P'\{w/y\} \mathrel{\mathcal{S}} Q'\{w/y\}$;
3. if $P \xrightarrow{\overline{x}(y)} P'$ and $y \notin n(P, Q)$, then there $\exists Q'.Q \xrightarrow{\overline{x}(y)} Q'$ and $P' \mathrel{\mathcal{S}} Q'$.

$\mathcal{S}$ is a *strong bisimulation* if both $\mathcal{S}$ and $\mathcal{S}^{-1}$ are strong simulations.
Strong bisimilarity ($\dot{\sim}$) is defined as

$$P \mathrel{\dot{\sim}} Q \iff \exists \mathcal{S} \text{ strong bisimulation.} (P \mathrel{\mathcal{S}} Q)$$

The system $\mathrm{CC}^{(Co)Ind}$/Coq (INRIA)

$\mathrm{CC}^{(Co)Ind}$ is an extension of the **Calculus of Constructions** [Coquand, Huet ('88)], featuring:

- (co)inductive types support [Giménez ('94)];
- functional languages typical constructs (case expressions, pattern matching à la ML, fixed-point operators);
- a very helpful proof tactic for developing proofs about coinductive predicates: the **Guarded Induction Principle**.

Coq is a proof assistant based on $CC^{(Co)Ind}$ featuring:

- a **logical metalanguage** for formal specifications;
- a **proof editing mode** for proof developments;
- a **program extractor**.

wHOAS-encoding of the π-calculus syntax

Names are represented by variables of type `name`,
processes are terms of type `proc`, free (bound) actions are terms of type `f_act` (`b_act`).

$\mathcal{N}, \mathcal{P}, \mathcal{L}$	$\rightsquigarrow$	`name,proc,f_act,b_act:Set`
0	$\rightsquigarrow$	`nil : proc`
$!$	$\rightsquigarrow$	`bang : proc -> proc`
τ	$\rightsquigarrow$	`tau_pref: proc -> proc`
$+, \mid$	$\rightsquigarrow$	`sum,par : proc -> proc -> proc`
ν	$\rightsquigarrow$	`nu : (name -> proc) -> proc`
$[\_ = \_]$	$\rightsquigarrow$	`match_ : name -> name -> proc -> proc`
$[\_ \neq \_]$	$\rightsquigarrow$	`mismatch : name -> name -> proc -> proc`
$\_(\_)$	$\rightsquigarrow$	`in_pref : name -> (name -> proc) -> proc`
$\bar{\_}\,\_$	$\rightsquigarrow$	`out_pref : name -> name -> proc -> proc`

$x(y).P \rightsquigarrow$ `(in_pref x (fun y:name =>` $\hat{P}$`)) : proc`

Remark: `proc`, `f_act`, `b_act` are inductive types, `name` is not.

wHOAS-encoding of the π-calculus LTS

The *Labelled Transition System* is represented by two mutually defined inductive predicates:

```
Inductive ftrans : proc -> f_act -> proc -> Prop := ...
with      btrans : proc -> b_act
                         -> (name -> proc) -> Prop := ...
```

(such encoding is inspired by an idea of J. Parrow: $P \xrightarrow{x(y)} P' \rightsquigarrow P \xrightarrow{x} \lambda y.P'$)
and four auxiliary predicates representing *freshness* of names:

```
Inductive       notin (x:name) : proc  -> Prop := ...
Inductive f_act_notin (x:name) : f_act -> Prop := ...
Inductive b_act_notin (x:name) : b_act -> Prop := ...
Inductive Nlist_notin (x:name) : Nlist -> Prop := ...
```

$$\mathrm{IN}\ \frac{-}{x(z).P \xrightarrow{x(w)} P\{w/z\}}\ w \notin fn((\nu z)P)$$

```
IN: forall (p: name -> proc) (x: name),
    btrans (in_pref x p) (In x) p
```

$$\mathrm{COM}_l\ \frac{P \xrightarrow{\overline{x}y} P' \quad Q \xrightarrow{x(z)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{y/z\}}$$

```
COM1: forall (p1 p2 q2: proc) (q1: name -> proc) (x y: name),
       btrans p1 (In x) q1
             -> ftrans p2 (Out x y) q2
      -> ftrans (par p1 p2) tau (par (q1 y) q2)
```

Side conditions are automatically dealth with by wHOAS.

Formal development of the metatheory

- We need to "reify" at the object level the notion of freshness of bound variables:

```
fRES: forall (p1 p2 : name -> proc) (a : f_act) (l : Nlist),
                (forall y : name, notin y (nu p1) -> notin y (nu p2)
                 -> Nlist_notin y l -> f_act_notin y a
                 -> ftrans (p1 y) a (p2 y)
                ) -> ftrans (nu p1) a (nu p2)
```

- The list `l` above is necessary in order to keep trace of possible names occurring in the environment where the process "lives", when such names do not occur in the process itself.
- Finally, we need to axiomatize some basic properties about names and contexts (i.e., terms with "holes").
  - Example: $P[\cdot] \equiv \overline{x} \cdot .0|\cdot(z).0$ is a context. Then, $P[y] \equiv \overline{x}y.0|y(z).0$ is the process obtained from $P[\cdot]$ filling in the name y.
  - In Coq contexts are rendered as functional terms. For instance, the above mentioned $P[\cdot]$ is represented by:

    ```
    fun y:name => par (out_pref x y nil) (in_pref y fun z:name => nil)
    ```

Properties of names and contexts: the Theory of Contexts

Unsaturation: $\forall P.\forall L.\exists x.x \notin P \wedge x \notin L$
Decidability of equality between names: $\forall x, y.\ x = y \ \vee \ x \neq y$
β-expansion: $\forall P.\forall x.\exists Q.P = Q[x] \wedge x \notin Q[\cdot]$
Extensionality: $\forall P.\forall Q.\forall x.x \notin P[\cdot], Q[\cdot] \Rightarrow P[x] = Q[x] \Rightarrow P[\cdot] = Q[\cdot]$

Sample encoding in `Coq` as `Axioms` (for type `proc`).

```
unsat      : forall (p : proc) (l : Nlist),
             exists x : name, notin x p /\ Nlist_notin x l.
LEM_name   : forall x y : name, x = y \/ x <> y.
proc_exp   : forall (p : proc) (x : name),
             exists q : name -> proc, notin x (nu q) /\ p = q x.
proc_ext   : forall (p q : name -> proc) (x : name),
             notin x (nu p) -> notin x (nu q) -> p x = q x -> p = q.
```

1. When there are no binders, some properties are derivable in `Coq`.
2. You may need higher-order versions of β-expansion and extensionality:

```
Axiom ho_proc_ext : forall (p q : name -> name -> proc) (x : name),
  notin x (nu (fun y : name => nu (p y))) ->
  notin x (nu (fun y : name => nu (q y))) -> p x = q x -> p = q.
```

Properties derived from the ToC

The following properties were originally included in the ToC, but later on they have been derived:

- Decidability of occurence checking: $\forall M.\forall x.x \in M \vee x \notin M$
- Monotonicity: $\forall M.\forall x.\forall y.x \notin M[y] \Rightarrow x \notin M[\cdot]$

Corresponding statements in `Coq` (for type `proc`).

```
LEM_OC    : forall (p : proc) (x : name), isin x p \/ notin x p.
proc_mono : forall (p : name -> proc) (x y : name),
            notin x (p y) -> notin x (nu p).
```

Deriving higher-order induction principles

An important result on the expressiveness of the Theory of Contexts is that it allows to derive (by means of a *complete induction* on the number of constructors contained in a term) higher-order induction principles.

```
Lemma HO_PROC_IND:
  forall P:(name->proc)->Prop,
  (P (fun x:name => nil)) ->
  (forall m:name->proc, P m -> P (fun x:name => bang (m x))) ->
  (forall m:name->proc, P m -> P (fun x:name => tau_pref (m x))) ->
  (forall m n:name->proc, (P m) -> (P m)
          -> (P (fun x:name => (par (m x) (n x))))) ->
  (forall m n:name->proc, (P m) -> (P m)
          -> (P (fun x:name => (sum (m x) (n x))))) ->
  (forall m:name->name->proc, (forall y:name, (P (fun x:name => m x y)))
          -> (P (fun x:name => (nu (m x))))) ->
  ...
  (forall m:name->proc, P m).
```

Deriving higher-order induction principles (cont.)

During the proof the axioms of β-expansion and extensionality is fundamental to "lift" structural information from `m:proc` to `n:name->proc` in order to apply the inductive hypothesis:

1. let us suppose that we are dealing with the case relative to parallel composition;
2. then, all we know is that `(m x)=(par P Q)` holds, for a variable `x` not occurring into `m`;
3. from `(m x)=(par P Q)`, we expand `P` and `Q` obtaining `P=(P' x)` and `Q=(Q' x)`;
4. then, from `(m x)=(par (P' x) (Q' x))`, we infer (by extensionality) that `m=(fun z:name => (par (P' z) (Q' z)))`;
5. at this point we have sufficient structural information about `m` to apply the appropriate hypothesis.

Some statistics [HMS01]

In 2001, with a Sun Enterprise Server 450 with two UltraSPARC processors at 300MHz, 256MB RAM, 513MB swap space, Coq V6.2, in native mode:

Number of proofs:	90 (all the metatheory of LTS and $\dot{\sim}$)
Size of source code:	$\sim$350 KB
Length of proofs:	maximum : $\sim$ 57KB (Lemma 3) average : $\sim$ 3.9KB minimum : 178Byte (Soundness)
Broadest proof tree:	42 main subgoals (associativity of $\|$)

Times of compilation	
Theory:	42.3 sec
Cross adequacy:	39 sec
Theory of contexts:	38 sec
Lemmas 1–6:	1h 2m 31sec
Metatheory:	1h 1m 19sec
Congruence of $\dot{\sim}$ w.r.t. !:	11m 26sec
Maximum memory consumption:	187MB

Today, with an Intel Core i7 4700MQ at 2.40 GHz, 16GB RAM, Coq V8.4pl6, the whole compilation takes a time of 2 m 39,593 sec.

Schematic derivations: renamings

A lot of the π-calculus metatheory is built on the following lemmas:

Lemma 3 If $P \longrightarrow P'$, then for all $y \notin fn(P)$ $P[y/x] \longrightarrow P'[y/x]$.

Lemma 6 If $P \dot{\sim} Q$ and $y \notin fn(P, Q)$, then $P[y/x] \dot{\sim} Q[y/x]$.

In Coq:

```
Lemma FTR_L3 :  forall (p q : name -> proc) (a : name -> f_act) (x : name),
           notin x (nu p) -> notin x (nu q) -> f_act_notin_ho x a ->
           ftrans (p x) (a x) (q x) -> forall y : name,
           notin y (nu p) -> notin y (nu q) -> f_act_notin_ho y a
           -> ftrans (p y) (a y) (q y).
Lemma L6 : forall (p q : name -> proc) (z : name),
          notin z (nu p) -> notin z (nu q) -> StBisim (p z) (q z) ->
          forall w : name, notin w (nu p) -> notin w (nu q)
          -> StBisim (p w) (q w).
```

Remarks:

- Both lemmas ensure the possibility of replacing a name in a given derivation with a “fresh” one (schematic derivation).
- The above mentioned lemmas are paradigmatic examples of metatheoretic properties requiring the axioms of the ToC.
- wHOAS and the ToC, according to our experience, are good tools to express and deal with such kind of properties.

Caveat: name cannot be inductive...

If `name` would be an inductive type, we would have two kind of problems:

1. we would lose the encoding adequacy because of exotic terms of type `proc`:
```
(nu (fun x:nat => (match x with O => nil
                               | (S n) => (par nil nil) end)))
```
2. the encoding would be inconsistent:

$$\mathtt{x} \stackrel{\mathrm{def}}{=} \mathtt{O} \qquad \mathtt{y} \stackrel{\mathrm{def}}{=} (\mathtt{S}\ \mathtt{O}) \qquad \mathtt{p} \stackrel{\mathrm{def}}{=} \mathtt{fun}\ \mathtt{z} : \mathtt{name} => \mathtt{nil}$$

```
q ≝ (fun z:name => match z with O => nil
                              | (S n) => (par nil nil) end)
```

whence, `(p x)=(p y)=(q x)=nil`, while `(q y)=(par nil nil)`. It would follow by extensionality that `nil=(par nil nil)` (absurd).

. . . and beware of the Axiom of Unique Choice

Proposition. The Axiom of Unique Choice (AC!):

$$\frac{\Gamma \vdash_\Sigma P : \tau_1 \to \tau_2 \to Prop}{\Gamma \vdash_\Sigma (\forall x{:}\tau_1.\exists y{:}\tau_2.(P\ x\ y) \wedge \forall z{:}\tau_2.(P\ x\ z) \Rightarrow y = z) \Rightarrow \exists f{:}\tau_1 \to \tau_2.\forall x{:}\tau_1.(P\ x\ (f\ x))}$$

is **inconsistent** with the Theory of Contexts.

Proof (sketch):

- $R \triangleq \lambda u : name.\lambda q : proc.\lambda x : name.\lambda p : proc.(x = u \wedge p = 0) \vee (\neg x = u \wedge p = q)$;
- *Unsaturation* allows to infer the existence of a fresh name u';
- it is easy to show that, for every p', $(R\ u'\ p') : name \to proc \to Prop$ is a functional binary relation;
- the proposition $\forall p' : proc.p' = 0$ holds:
  - AC! allows to infer the existence of a function $f : name \to proc$ s.t., for all $x : name$, $((R\ u'\ p')\ x\ (f\ x))$ holds;
  - by *Extensionality*, we have that $f = \lambda x : name.p'$ because we have $(f\ w) = p'$ (for any fresh w);
  - then, for all $y : name$, $(f\ y) = ((\lambda x : name.p')\ y) = p'$ holds, whence the thesis, since $(f\ u') = 0$;
- the contradiction follows since, as a special case, we have that $0|0 = 0$.

Remark: if we assume `Set=Prop`, then AC! is derivable in Coq.

. . . but then. . . all those axioms. . . Are they safe?

- Yes, the axioms of the ToC are consistent with the type theory of `Coq`.
- In particular, there is a categorical model based on an idea of M. Hofmann:

  A. Bucalo, M. Hofmann, F. Honsell, M. Miculan, I. Scagnetto
  Consistency of the Theory of Contexts.
  Journal of Functional Programming, Volume 16, Issue 03, May 2006, pp 327-395.

- In Isabelle/HOL the ToC is completely derivable:

  C. Röckl, D. Hirschkoff, S. Berghofer.
  Higher-Order Abstract Syntax with Induction in Isabelle/HOL: Formalizing the Pi-Calculus and Mechanizing the Theory of Contexts.
  In Proc. FOSSACS'01, LNCS 2030. Springer, 2001.

- Another interesting application scenario for the ToC: The POPLmark Challenge (Parts 1A-1B) [CS15].

A little bit of advertising

- If you are working on a formalization of a process algebra (even if is at the "work in progress" stage), and you want to discuss it with an expert community, please consider to submit your work to

  Logical Frameworks and Meta-Languages: Theory and Practice
  LFMTP 2019

  Vancouver, CA, 22 June 2019
  Affiliated with LICS 2019

  `http://lfmtp.org/workshops/2019/`

- **Extended deadline:** April 15^{th}

Arthur Charguéraud.
The locally nameless representation.
Journal of Automated Reasoning, 49(3):363–408, Oct 2012.

Alberto Ciaffaglione and Ivan Scagnetto.
Mechanizing type environments in weak hoas.
Theoretical Computer Science, 606:57 – 78, 2015.
Logical and Semantic Frameworks with Applications.

Nicolaas Govert De Bruijn.
Lambda Calculus Notation with Nameless Dummies: A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem.
Indagationes Mathematicae, 34:381–392, 1972.

Joëlle Despeyroux, Amy Felty, and André Hirschowitz.
Higher-order abstract syntax in coq.
In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 124–138, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

Andrew D. Gordon and Tom Melham.
Five axioms of alpha-conversion.

In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics*, pages 173–190, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

Murdoch Gabbay and Andrew Pitts.
A new approach to abstract syntax involving binders.
In *Logic in Computer Science, 1999. Proceedings. 14th Symposium on*, pages 214–224. IEEE, 1999.

André Hirschowitz and Marco Maggesi.
Nested abstract syntax in coq.
Journal of Automated Reasoning, 49(3):409–426, Oct 2012.

F. Honsell, M. Miculan, and I. Scagnetto.
π-calculus in (Co)Inductive Type Theories.
Theoretical Computer Science, 253(2):239–285, 2001.

Robin Milner, Joachim Parrow, and David Walker.
A calculus of mobile processes, i.
Information and Computation, 100(1):1 – 40, 1992.

Robin Milner, Joachim Parrow, and David Walker.
A calculus of mobile processes, ii.
Information and Computation, 100(1):41 – 77, 1992.

F. Pfenning and C. Elliott.
Higher-order abstract syntax.
SIGPLAN Not., 23(7):199–208, June 1988.